AD A118286

THE TOOLPACK MATHEMATICAL SOFTWARE DEVELOPMENT ENVIRONMENT

by

Leon J. Osterweil

Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-226-82 July 21, 1982

This research was supported, in part, by NSF grant MCS MSC8000017, Department of Energy grant number DEAC02-80ER10718, and U.S. Army Research Office Contract number DAAG 29-80-C-0094.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A118 286	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE TOOLPACK MATHEMATICAL SOFTWARE DEVELOPMENT ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) L. J. OSTERWEIL		8. CONTRACT OR GRANT NUMBER(s) DAAG29 80 C 0094
9. PERFORMING ORGANIZATION NAME AND ADDRESS UNIVERSITY OF COLORADO BOULDER, CO 80309		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE JUL 82
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE NA

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software tools, environments, mathematical software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a research project aimed at building and studying prototype environments for Mathematical Software. The actual measurements and experiences that should help solidify knowledge about how to build effective environments. Towards this end some speculative ideas about environment file systems and command languages are presented, along with research plans for effectively evaluating these and other design notions.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Introduction

This paper describes Toolpack[1], a project still in progress to build a prototype software development environment. The purpose of the Toolpack project is to gain insight into some of the central questions confronting the would-be builder of a software environment by creating and studying an experimental prototype.

It is becoming generally agreed that if software tools are to be effectively exploited they must be amalgamated into well integrated collections. It also seems agreed that these collections should be broad in scope, easy to use, and highly efficient in actual operation. Such a well integrated tool collection has come to be called an environment. To be more specific, in an earlier paper [Oste 81] it was proposed that an environment was a collection of software tools which had the following five properties:

(1) Breadth of scope--capabilities spanning the entire range of activities to be performed in order to accomplish a complete specific software job.

(2) User Friendliness--Input language and diagnostic capabilities which would neither intimidate nor harass the user, as well as sufficient adaptability to assure that the tools would remain useful and supportive as the user's work procedures and style underwent reasonable changes.

(3) Tight Integration--Tools which are sufficiently aware of each other's capabilities to avoid the semblance of overlapping capabilities as well as the possibility of incompatibility.

(4) Internal Reusability--An architecture and design which encourages the reuse of simple modular capabilities in furnishing the various functional capabilities of the environment.

[1]Toolpack is a research project involving cooperation among researchers from Argonne National Laboratory, Bell Telephone Laboratories, IMSL Inc., Jet Propulsion Laboratory, Numerical Algorithms Group Ltd., Purdue University, University of Arizona and University of Colorado at Boulder. The project is supported by the National Science Foundation, the Department of Energy and the participating institutions, themselves.

(5) Use of a Central Data Base--an architecture and design in which the various functional tools draw their inputs from, and place their outputs back into, a central repository of information. This repository is to be considered the focus of all knowledge about the software project.

Although much of this seems clear and well agreed to, it is far less clear how to go about building such an environment. In particular it is unclear how to achieve each of the five characteristics just described with a single software system. There is even considerable question about whether the five are consistent and compatible with each other.

For example, it has been suggested [Oste 81] that tight integration and internal reusability might be inconsistent objectives. It appears, at least on the surface, that tools which are keenly aware of each other might be difficult or impossible to construct from standard self-contained modules.

The need to center the environment around a data base containing all project information also poses a problem. Clearly a given software development project generates an enormous amount of information. If the data base is to contain all information spanning all aspects and phases of the project then it would have to contain all of this information and reflect all of the myriad relations which characterize the project and its status. It is not clear that this can be done in an efficient, cost-effective way. Thus there arises the question of whether a central data base can be created in a manner which is consistent with the need for efficiency and breadth of scope.

In considering how such an environment might be built it is reasonable to look to the paradigm of the software development lifecycle as a model, and to simply follow accepted software development lifecycle procedure. Thus conventional wisdom suggests that one should start construction of a software development environment by carrying out a careful analysis of the requirements. In the process of doing so one should obtain the answers to the questions raised above. For example, careful requirements analysis should make clear the specific actual needs for the various pieces of information and relations which must be in the central data base. Similarly the requirements analysis process should make clear the performance requirements (e.g., access speeds) necessary in order for the environment to be acceptably fast and inexpensive.

Here the circular nature of this problem starts to become apparent. In order to pinpoint the requirements for an effective software development environment sufficiently to definitively obtain answers to the above questions, it is essential to be able to interview a wide variety of software developers who are knowledgeable and experienced in such matters. Specifically, it is essential to get definitive answers about experiences and judgments concerning specific tool capabilities and the items of information which they utilize and create. It is unfortunately the case that this sort of knowledge and experience is very rare, because of the lack of widespread use of a variety of software development tools. In fact it is the lack of widespread effective utilization of superior tools that has led us to believe that environments must be created. Hence there seems to be a paradox in that the knowledge needed to form the basis for an environment building effort is not available for precisely the same reasons that are prompting the effort in the first place.

Furthermore, there seems to be agreement that access to superior software development environments will rapidly cause developers to change the manner in which they do their work. Thus any guesses about what might be needed in the way of tool capabilities and configurations would probably change as experience with environments grew. Thus it seems that here, as in the case of other software projects which are designed to address a new and evolving problem, it is naive to expect that we will be able to definitively establish a firm baseline set of requirements.

Instead what must be done is to evolve a strategy for studying the requirements for a software development environment which assures that there is steady progress towards the goal of sufficient knowledge and confidence to justify embarking upon a full scale environment development activity. One way of doing this is to embark on a program of constructing a series of increasingly ambitious experimental prototype environments. If each prototype is designed to be the object of study and the sequence is arranged in such a way that the most critical requirements and design issues can be elucidated or resolved by the early prototypes, then there would seem to be good reason to expect that an effective environment would emerge as an end-product of this process.

It is in this spirit that the Toolpack project has embarked upon a plan for producing a sequence of at least three successive releases of an environment for software production.

2. Goals of the Toolpack Prototype Development Effort

The main goal of the Toolpack project to is establish a positive feedback loop between environment developers and a broad and diverse base of environment users by supplying those users a sequence of environments that is increasingly responsive to the users' needs. The purpose of this feedback process is twofold. One purpose is to create and promulgate a vehicle for the more effective development and maintenance of software. The other is to obtain reliable, detailed, quantitative answers to many of the central questions confronting software development environment builders. Specifically, it is expected that at least the following issues will be elucidated or resolved:

(1) What is an acceptably broad and complete suite of tool capabilities for supporting some specific software development jobs?

(2) How important are various data items and relations and how accessible must they be to various sorts of environment users?

(3) Is there a set of modular "tool fragments" which is sufficiently powerful yet flexible to provide the basis for a broad yet tightly integrated set of tool capabilities?

(4) What are the general characteristics of a user interface language which is sufficiently powerful, yet acceptably "friendly"?

In order to get reliable insights into these questions it is important to construct prototypes in such a way that a large and diverse community of users will become active users of the environment. Thus the Toolpack project has taken great pains to assure that its prototype environments will be of great interest and assistance to a large and significant community of users.

The target community for the Toolpack prototypes is the community of Mathematical Software developers. This community is in many ways a nearly ideal target community. The mathematical software community is among the oldest software production communities, tracing its origins to the small group of scientists who conceived of the stored program computer in the 1940's. Thus it is a coherent group that has well agreed upon goals and procedures. Among the accepted procedures of this community is the utilization of tools (e.g., see [JPL 78], [JPL 81]). In addition, the community has consciously and innovatively striven toward quality for perhaps a longer period of time than any other software community. This has manifested itself for example, in the notable "PACK" projects of the 1970's [Cowe 77].

This community is perhaps unique in that it has long held that portability is a necessary characteristic of high quality software. Thus the community is accustomed to receiving and evaluating new software items as a community, regardless of differences in the hardware/software configurations being used by different members of the community. Of course it is essential that such new software items be presented in portable form.

There are other characteristics of the mathematical software community that cause it to be very desirable for our purposes. It has long ago established a single programming language (Fortran) as its, more or less uniform, standard. Thus a suite of support tools of interest and value to the entire community can be made source language specific. This greatly simplifies the problem of writing generally useful and acceptable tools.

The software produced by this community is ordinarily rather modest in size, usually aggregating less than 10,000 lines of source code. This also simplifies the problem of writing acceptably efficient tools.

In addition, the mathematical software community generally follows a software lifecycle model which is far simpler than the lifecycle models which are widely espoused by and for many other software development communities. Mathematical software development rarely, if ever, begins with formal requirements analysis. Similarly, there is rarely a formal preliminary (or architectural) design phase. This appears at first glance to be paradoxical, especially in view of the high quality and good acceptance of mathematical software over the past decades. The explanation appears to be that mathematical software requirements and preliminary design specifications have been derived over a period of decades (if not centuries) by mathematicians and numerical analysts. These specifications appear as mathematical formulas in books and technical reports. Thus the process of producing mathematical software appears to start on this base and proceed immediately with what other communities would label detailed (or algorithmic) design. These designs are often expressed directly in the form of code for a higher level pseudolanguage such as SFTRAN [JPL 81a], Ratfor [Kern 75] or EFL [Feld 79], but perhaps more routinely they are coded directly in a relatively portable Fortran dialect. There then follows a familiar pattern of testing, documentation and the upgrading and adjustment that is most often referred to as "maintenance".

It is doubtlessly true that mathematical software began encountering, and grappling with, the sorts of data manipulation problems whose solution would benefit from the more formal requirements and preliminary design techniques prevalent in other communities today. It might be interesting to conjecture about why these contemporary techniques have never been adopted by the mathematical software community, but such conjecture would digress from the issue at hand. The issue is that this community currently does not generally perceive the need for these techniques and associated tools. Thus their absence from Toolpack should not endanger community acceptance. This acceptance can be based only on solid support for the lifecycle as it is practiced, rather than as it might or should be practiced.

This prevailing lifecycle model is particularly fortunate for the Toolpack project because is suggests that a tool support set can be relatively modest, addressing the creation, testing, analysis, documentation, and transportation of only code (and perhaps some algorithmic design) and still be considered to be a complete tool set by this community. More fortunately still, most if not all of these tool capabilities have already been produced and evaluated to some extent by some members of the community. Thus a comprehensive tool set will not be totally unfamiliar to the community. Further, the preexistence of such tools means that the environment production activity can focus more on the issues of integration, user interface, and data base contents, and will not need to be preoccupied with more mundane matters such as the recreation of tool capabilities whose reproduction will contribute less to the accumulation of new knowledge about environments.

With these factors in mind, the Toolpack project has set out to build a portable environment capable of extending comprehensive support to the community of people who are engaged in producing, testing, transporting and analyzing mathematical software written in Fortran. Toolpack project environments will be made available through a series of releases, each of which is designed to improve upon its predecessors as a consequence of experience and evaluation obtained through extensive and diverse utilization of those predecessors.

The specific approach to the architecture and design of the family of Toolpack environments will now be summarized. A more complete summary can be found in [Oste 81a]. It should be stressed that this approach has been arrived at only after extensive discussions and consultations designed to determine the preferences and predispositions of the mathematical software community so as to assure, as well as feasible, widespread acceptance of the Toolpack project environment releases. The approach has also been designed so as to assure that the qualitative and quantitative observations of user experiences will make substantial contributions towards resolution of the critical environment design issues enumerated above.

3. Requirements and Functional Capabilities of the Toolpack Environment

The purpose of Toolpack is to provide strong, comprehensive tool support to programmers who are producing, testing, transporting or analyzing moderate size mathematical software written in Fortran77.

3.1. Overall Requirements

The following are taken to be the basic assumptions upon which the Toolpack environment architecture and design are based.

(1) The mathematical software whose production, testing, transportation and analysis is to be supported by Toolpack systems shall be written in a dialect of Fortran 77. This dialect shall be carefully chosen to span the needs of as broad and numerous a user community as is practical.

(2) Toolpack software and systems are to be designed to provide cost effective support for the production by up to 3 programmers of programs whose length is up to 5ØØØ lines of source text. They may be less effective in supporting larger projects.

(3) Toolpack software and systems are to be designed to provide cost effective support for the analysis and transporting of programs whose length is up to 1Ø,ØØØ lines of source text. They may be less effective in supporting larger projects.

(4) Toolpack software and systems will support users working in either batch or interactive mode, but may offer stronger more flexible support to interactive users.

(5) Toolpack software and systems will be highly portable, making only weak assumptions about their operating environment. They will be designed, however, to make effective use of large amounts of primary and secondary memory, whenever these resources can be made available.

3.2. Toolpack Tools

The Toolpack group is in agreement that the following tool capabilities constitute a sound basis for a programming support system for the production of high quality Fortran programs:

Ø. A compiling/loading system
1. A Fortran-intelligent editor
2. A formatter
3. A structurer
4. A dynamic testing and validation aid
5. A dynamic debugging aid
6. A static error detection and validation aid
7. A static portability checking aid
8. A documentation generation aid
9. A program transformer

A compiling/loading capability is generally available on host operating systems. Thus no tool development effort in this area is proposed. Tools are to be developed in all of the nine other areas, however. In fact, significant tool capabilities have already been developed in some of these areas, as shall be described subsequently.

3.2.1. Fortran-Intelligent Editor

A powerful editor [Hagu 81] will be included in Toolpack to assist the programmer in producing Fortran source code. This editor will offer a range of general text manipulation facilities, including the usual capabilities for inserting, deleting, locating and transforming arbitrary strings of characters. In addition, the editor will provide the following facilities for constructing and modifying Fortran programs.

--The user will be able to abbreviate Fortran keywords; these abbreviations will be automatically expanded by the editor.

--The editor will assure that various fields of Fortran statements are placed in the proper columns.

--As with the Cornell Program Synthesizer [Teit 81] and the Mentor [Donz 8Ø] system, the Toolpack editor will prompt the user for anticipated constructs. Moreover, the subsequent incoming statement will be checked for syntactic correctness and certain kinds of semantic consistency, e.g., the usage of a variable against declarations of the variable.

--The user will be able to search for occurrences of specified variables and labels. The editor will be able to distinguish those occurrence from occurrences of the same string of characters in other contexts, e.g., comments.

--It will be possible to confine searching and replacement operations to fixed domains of a program, such as a particular DO loop or a particular subroutine. For example, it will be possible to change all occurrences of a given variable (say X) to another variable (say Y) within a specified subroutine.

3.2.2. Formatter

The Toolpack project will provide a tool to put Fortran programs into a canonical form. In particular, the formatting tool, called Polish-X [Fosd 81] will have the following capabilities.

--Variables and operators will be set off by exactly one space on either side, except in certain cases, e.g., subscripts.

--DO loop bodies and IF statement alternatives will be indented.

--Statement labels will optionally be put in regular increasing order.

--It will be possible to optionally align the lefthand and righthand margins of statements.

--It will be possible to insert ON and OFF markers to indicate that Polish-X is to leave certain sections of the program unaltered.

3.2.3. Structurer

The ability to infer and emphasize the underlying looping structure of a program is useful. The failure of Fortran 77 to supply suitable constructs for doing so has left a significant void in the language. Hence a tool is to be provided that will recast Fortran 77 program loops as, for example, DO WHILE loops, either simulated in Fortran 77 by canonical constructs or realized explicitly according to the rules of Ratfor [Kern 75], EFL [Feld 79] or SFTRAN [JPL 81a]. This tool will, moreover, be able to automatically upgrade many Fortran 66 GO TO's to Fortran 77 IF-THEN-ELSE constructs. Such structuring often improves readability and comprehensibility, and serves as valuable documentation. The structuring capability in Toolpack will be closely patterned after the UNIX[2] struct command [Bake 77].

3.2.4. Dynamic Testing and Validation Aid

The Toolpack project will produce a facility for automatically inserting instrumentation probes into Fortran 77 programs and for creating useful intermediate output from these probes at run time. This facility will enable the user to capture and view a variety of trace and summary information. It will be possible, for example, to capture a program's statement execution sequence or to generate a histogram of the relative frequencies of execution of the various statements. Similarly, it will be possible to capture and study subroutine execution sequences and histograms, or variable evolution histories.

It will be possible for the user to implant in the subject program monitors for certain kinds of errors. For example, the user will be able to specify that either all or certain specified arrays are to be monitored to be sure that the subscripts by which they are referenced stay within declared bounds.

This facility will also incorporate a capability for checking the outcome of an execution against specifications of intent fashioned by the user. The specifications of intent are to be embodied in assertions, stated in comments and expressed in a flexible assertion language. These assertions will be expanded into executable code by the Toolpack dynamic testing facility. Once an assertion violation is detected, relevant information about the program status at the time of the violation will be automatically saved.

A system, call Newton [Feib 81] (see Figure 1), is being developed to provide the functional capabilities just outlined.

The dynamic testing capability will make it all to easy to produce very large amounts of output. There is some sentiment among Toolpack group members that tool support should be provided to aid the process of inferring useful diagnostic and documentation information from this raw output. This support would treat the output from the dynamic execution as a data base of information, and would consist of data base management aids and report generators to organize and format the diagnostic output for ease of understanding. There are currently no firm plans to construct such a tool.

[2] UNIX is a trademark of Bell Laboratories.

3.2.5. Dynamic Debugging Aid

Debugging is to be facilitated by the ability to scrutinize to arbitrary levels of details the progress of the execution of a program that is behaving incorrectly. Thus the Newton system, classified above as a dynamic testing and validation aid, can be viewed as a debugging aid as well.

In addition, however, debugging is assisted by snapshot, breakpoint and single-step-execution capabilities. These are also to be provided by Newton. They will enable a user to suspend execution at designated sites or on designated conditions. While execution is suspended the user will be able to examine the current values of variables, the execution history to date and the source text.

3.2.6. Static Error Detection and Validation Aid

The Toolpack project will furnish flexible capabilities for statically detecting a wide range of errors and, where possible, for proving the absence of errors. These tools will offer the user the ability to easily select from a range of capabilities that includes those supplied by the Dave data flow analysis system [Oste 76].

The structures of modern modular compilers and of the Dave II system suggest that the static analysis of a program can be organized into the following progression of analytic steps: lexical analysis, syntactic analysis, static semantic analysis and data flow analysis. Thus the Toolpack static analysis capability will be subdivided into individually selectable capabilities offering these levels of analytic power (see Figure 2).

The lexical analysis step will accept as input the program source text, and convert it into the corresponding list of lexical tokens. Illegal tokens such as unknown keywords or variables that are too long, will be detected in this process and reported.

The syntactic analysis step will require the list of lexical tokens as input. This process will construct a parse tree representation of the user's program and a symbol table. In the process of doing this, syntactic errors, such as illegal expressions or malformed statements, will be detected and reported.

The static semantic analysis step will build upon the output of the first two static analyzers and will produce a number of structures designed to represent and elucidate the functioning of the program. These structures will facilitate the checking and cross-checking that can detect such errors as mismatched argument and parameter lists, unreachable code segments, inconsistencies between variable declaration and usage, and improper DO loop nesting and specification.

The data flow analysis step will rest upon the semantic information and flowgraph structures built by the other three analyzers, and will produce reports about the references and definitions affected by each statement and subprogram of the usrer's program. These reports will then be the basis for analytic scans of all possible program execution sequences. These scans will produce reports about whether there is any possibility of referencing a program variable before it has been defined, or defining a program variable and then never referencing it.

There is some sentiment among Toolpack group members that a tool is needed for centralizing and coordinating error reporting from these four static analysis tools. Such a tool would be similar in purpose to the error reporting tool discussed in Section 3.4. Here too, there is currently no firm plan to build such a tool.

3.2.7. Static Portability Checking Aid

The Toolpack project will furnish a capability for statically determining whether or not a given Fortran 77 program is written in such a dialect and style as to facilitate transporting the program. This capability will be modeled after the PFORT Verifier [Ryde 74], a very successful and useful tool for checking the portability of Fortran 66 programs. Such portability obstacles as use of statement types not defined in the language standard (e.g., NAMELIST), assumptions about word lengths (e.g., packing of multiple characters in a word without use of the CHARACTER data type), and use of non-portable machine constants will be detected and reported.

Certain interprocedural checks not done by the PFORT Verifier, but supported by Dave, will be incorporated into the Toolpack portability checker. For example, Fortran programs sometimes rely for correct execution upon assumptions about the parameter passing mechanism of the compiler on which the programs were developed. Data flow analysis determines the treatment of every parameter and COMMON variable by every subprogram with sufficient precision that non-portable parameter passing practices can be detected.

The functional capabilities needed to create this portability aid are quite similar to those needed by the static error detection and validation aid just described. A primary difference is that this tool, as opposed to most other

Toolpack tools, will need to analyze and support a restricted Fortran 77 dialect, as opposed to a liberalized, extended dialect.

3.2.8. Documentation Generation Aid

The Toolpack group recognizes the importance of high quality program documentation and the desirability of tool support for the process of creating it. The group believes that the static and dynamic analysis capabilities already described create items of information that are useful program documentation. A documentation aid might well draw upon this information and facilitate its availability. In addition, the documentation aid might assist the preparation of user-generated documentation. No specification for this tool is currently available.

3.2.9. Program Transformer

There is general agreement among Toolpack group members that it is highly desirable to produce a program transformation tool as part of the Toolpack project. It is agreed that the tool should offer such capabilities as assistance in translating one dialect of Fortran to another, assistance in altering a program to perform its computations at a different level of precision, and facilities for creating special-purpose control or data structures.

There is currently little agreement, however, about the tradeoffs this tool should make between power, rigor, efficiency and usability. Three specific tools have been proposed--a template processor system, a macro processor system, and a correctness-preserving transformation system.

The template processor [Ward 81] is designed to enable the user to define Fortran language extensions by establishing data structure "templates". These templates can be named in the body of a Fortran program along with program data objects. The program data objects are then taken as arguments to be imbedded in the template description. The effect of this is that the user can employ and manipulate complex data structures in a source program without having to define those data structures within the program. This also leaves open the possibility that an expert could establish these complex data structure templates for users who lack the expertise to create the structures, but who, nevertheless, have a need for them. The template processor is designed to be extremely easy to use, but does little to guarantee that the Fortran statements it generates are correct or efficient.

The macro processor, called BIGMAC II [Myer 81], is similar in operation to the template processor. It enables users to define macros (code skeletons) which can then be expanded into actual bodies of code with the incorporation of parameters supplied to the macro at an invocation site in a user's program. Here too, the macros can be written by an expert and made available to casual users, much like a library subprogram. BIGMAC II macros are more complicated and difficult to write than templates, but have the advantage of assisting the writer in preparing efficient and correct Fortran programs. These differences spring, essentially, from the ability of BIGMAC II macros to acquire information about their invoking environments, to communicate with each other, and to produce output Fortran code that can be implanted in a few different strategic places in the source code of the invoking program.

The correctness-preserving transformation system, called TAMPR [Boyl 76], is the most powerful and sophisticated of the three proposed transformation systems. TAMPR constructs a parse-tree representation of the subject Fortran program, enables the user to analyze and transform the tree, and finally translates the transformed tree back to equivalent Fortran source code. TAMPR scrutinizes the transformation rules to be sure that the transformations that they specify do not alter the functionality of the subject program. This aspect of TAMPR makes it the safest of the three transformation systems. In addition, because the user is completely free to analyze and transform as much of the tree as desired, TAMPR has virtually limitless transformation power. The main drawbacks to this system are that, at least in prototype form, it appears to be very expensive to use and requires that the user be highly skilled and conversant with mathematical formalism.

In order to assist the Toolpack group in evaluating these three alternatives, it is likely that all will be made available as part of Toolpack so that a large and diverse user community can compare and evaluate them in a variety of usage contexts.

3.2.10. Additional Capabilities

Support from individual Toolpack group members has been expressed for eventual inclusion of a preprocessor for Ratfor [Kern 75], EFL [Feld 79], or SFTRAN [JPL 81a], for a document preparation aid like ROFF, for a source text version control facility, for a tape archiving program and for a general- purpose macro processor as advocated in [Mill 82]. Decisions about inclusion of such capabilities in Toolpack will hinge upon perceived user demand.

4. Tool Integration Strategy

The tool objectives described in Section 3 are to be achieved by a software system, currently implemented in prototype form, called the Integrated System of Tools (IST). A primary motivating goal of the architecture and design of the IST is that user support be supplied in as direct and painless a fashion as is feasible. In particular, the IST attempts to relieve the user of having to understand the natures and idiosyncrasies of individual Toolpack tools. It also relieves the user of the burden of having to combine or coordinate these tools. Instead the IST encourages the user to express needs in terms of the requirements of the actual software job. The IST is designed to then ascertain which tools are necessary, properly configure those tools, and present the results of using the tools to the user in a convenient form.

The architecture and design encourage the user to think of the IST as an energetic, reasonably bright assistant, capable of answering questions, performing menial but onerous tasks and storing and retrieving important bodies of data.

In order to reach this view, the user should think of IST as a vehicle for establishing and maintaining a file system containing all information important to the user, and using that file system to both furnish input to needed tools and capture the output of those tools. Clearly, such a file system is potentially quite large and is to contain a diversity of stored entities. Source code modules would certainly reside in the file system, but so would such more arcane entities as token lists, and flowgraph annotations. In order to keep IST's user image as straightforward as possible this design proposes that most file system management be done automatically and internally to the IST, out of the sight and sphere of responsibility of the user. The user, in addition is to be encouraged to have access to only a relatively small number of files - only those such as source code modules and test data sets which are of direct concern. The user may create, delete, alter and rename these entities. More important, however, the user may manipulate these entities with a set of commands which selectively and automatically configure and actuate the Toolpack tool ensemble. The commands are designed to be easy to understand and use. They borrow heavily on the terminology used by a programmer in creating and testing code, and conceal the sometimes considerable tool mechanisms needed to effect the results desired by the user.

4.1. User Visible IST File System Entities

In order to encourage and facilitate the preceding view, IST will support the naming, storage, retrieval, editing and manipulation of the following classes of entities, which should be considered to be the basic objects of IST:

4.1.1. Program units

An IST program unit (PU) is the same as a Fortran program unit, except that IST will require a number of representations of the program unit other than the source code (e.g., the corresponding token list and parse tree). The identity, significance, and utilization of these other representations are to be made transparent to the casual user. They will be managed automatically by IST. On the other hand, they will be accessible and usable by more expert users through published standard naming conventions and accessing functions.

4.1.2. Program Unit Groups

Any set of IST program units which the user chooses to designate, can be grouped into an IST program unit group (PUG). Other PUG's may also be named as constituents of a PUG, as long as no circularity is implied by such definitions. Ordinarily it is expected that a PUG will be a body of code which is to be tested as part of the incremental construction process. Hence a PUG might be a set of newly coded program units and a test harness. It is, however, not unreasonable (and indeed potentially quite useful) to consider a subprogram library to be a PUG as well. Here, too, an IST PUG will consist of more than just source text, but the user will not need to be aware of the existence of any such additional entities.

A PUG may also include optional transformation specifications which enable users to painlessly apply canonical transformations to their code. This will facilitate such functions as porting of code and coding in higher level pseudolanguages and languages such as Ratfor [Kern75], EFL [Feld79], and SFTRAN [JPL 81a].

4.1.3. Test Data Collections

An IST test data collection (TDC) is a collection of test data sets to be used in exercising one or more IST execution units. A test data collection may consist of one or more sets of the complete input data needed to drive the execution of some complete executable program. Each test input data set may also have associated with it a specification of the output which is expected in response to processing of the specified input.

4.1.4. Options Packets

An IST tool options packet (OP) is a set of directives specifying which of the many anticipated options are to be in force for a particular invocation of one of the Toolpack tools integrated into IST. We see, for example, the need for Test Option Packets (TOP's) to specify dynamic testing probe insertion options and Formatter Option Packets (FOP's) to specify program source text formatting options, among others. It is expected that some standard options packets will be created initially by the individual toolmakers and automatically incorporated as part of a newly installed IST. These standard options packets will undoubtedly be altered to meet the needs of individual users and installations. In addition, entirely new option packets will probably be built to satisfy individual needs. It should be stressed, however, that tool options will also be specifiable directly as part of a tool invocation command. Options so specified may either replace or supplement an option packet specification.

4.1.5. Procedures

An IST procedure is a sequence of IST commands which can be directed to the command interpreter simply by specifying the procedure name. IST procedures are expected to be command sequences for accomplishing generally useful standard jobs. Thus writing a procedure enables the user to save the effort of respecifying a standard sequence of commands whenever a standard job must be done. This capability is supplied as a convenience and is intended to supplement, not replace, the one-at-a-time command invocation capability. IST procedures will allow parameterization of inputs.

It is expected that Toolpack project software will facilitate the process of capturing and analyzing the procedures which users define and utilize. This will provide insight into the development procedures which users actually follow, as well as insight into how these procedures change with the availability of effective tools.

4.2. The File System

Clearly the primary feature of the proposed IST is the central file system of information about the subject program. The user is encouraged to think and plan work in terms of it, and the functional tools all draw their input from it and place their output into it. A schematic diagram of this architectural feature is shown in Figure 3.

IST itself will manage the file system primarily by means of a tree structured directory system and a modular set of file accessing and updating primitives. IST files will not correspond directly to host machine files,but will rather be mapped onto segments of one or more large host system files. The IST file accessing and updating capabilities will effect this segmentation and operate directly upon these large host system files. The objectives of this approach are to reduce the overhead of dealing directly with, and depending too heavily upon, host file systems and to increase the portability of IST. An implementation of such a set of I/O capabilities (called PIOS), has been written in portable Fortran [Hans 80b]. A tree structured file directory system (PDS) has also been written in portable Fortran [Hans80b]. These are quite appealing both as models of effective functional modularization and as actual implemented support libraries. They offer the added feature of being designed for ready interfacing with each other, thereby forming a portable file directory and accessing mechanism. This tandem has been used in implementing the IST file system in the first IST releases.

The IST file system is to be initialized with the start of a programming project and remain and grow throughout the lifetime of the project. There is no reason why several users may not all access this file system although PIOS requires that the file system be accessed by one user at a time, or by more than one user only in non-interfering ways.

This pragmatic restriction appears to be a workable one for this prototype effort. In the long run, however it threatens to be a severe one. A full-fledged multiuser file system offering file protection and permission capabilities will be needed if the Toolpack concepts are to be broadened to support wider needs and communities.

In contemplating these needs and appropriate solutions, one is lead, it seems inescapably, to the conclusion that such a file system must be viewed as the basis for effective configuration management and control. It would seem thus that it is only through this perspective that adequate requirements and consequent designs can be evolved.

The architecture of the IST anticipates the need for this in that the file system is designed and implemented as a separable module. The file system is directly used only by the IST command interpreter, and there only through standard functions and subroutines. In order to substitute a new file system it would be necessary only to have that file system support these existing functions and subroutines. It is true, of course, that a new file system incorporating protection and permission capabilities would require the submission of identification, such as passwords, with each file system access request. This requirement need not necessitate the alteration of existing calling sequences, however, as this identification could (perhaps should) reside in the user's global data areas.

Perhaps the most striking aspect of the IST architecture is the fact that it does not hypothesize a relational data base as its central element but rather a file system. As noted earlier, certain elements of the file system (e.g., the outputs of the static and dynamic analysis tool capabilities) seem to be best thought of as relational data bases. It is less clear that it is essential for the entire Toolpack information repository to be relational. It is important that this hypothesis be tested and evaluated because the operational costs of large relational data base systems appear to be quite high, possibly undermining their practical utility.

Toolpack will test the important hypothesis that a software development environment can be successful even when relational data base technology is applied only to smaller, more localized bodies of data.

In order for these objectives to be achieved there must be an underlying agreement about the naming of file system entities. As stated earlier, each of the file system entities created by the user (see Section 4.1) is to have a unique name which the user assigns. Different views, versions, or aspects of the entity are to have distinct names which are to be arrived at by attaching qualifiers as dictated by the published naming conventions to the user name. In some modes of use, the user will not need to be aware of these qualified names in order to get useful work done. This is because the IST will have considerable power to infer the names of needed views and versions from the context of the IST commands issued by the user. More advanced users seeking to carry out the more powerful and sophisticated functions of the IST will find it very important to know these naming and qualification conventions, however. In order to best understand this the IST command language will be presented next.

4.3. The IST Command Language

The form of an IST command will always be as follows:

```
command_name   list_of_pu's    options_specification
```

where the command_name must be chosen from the list of available tool capabilities, list_of_pu's is a list of PU's and/or PUG's which the user specifies, and options_specification is either the name of an options packet for the tool specified, or an explicit list of options specifications, or both.

In the current prototype version of the IST, the command names have been defined as two letter sequences because it was believed that users would prefer to avoid verbosity. Thus in order to invoke the formatting tool, the user would input the sequence "fm". In order to invoke the static analysis capability, the user would input the sequence "an", and so forth. Actual user experience with these choices and user reactions to them will dictate whether or not a more verbose form of these command names will eventually be adopted.

The list_of_pu's which follows the command name is the list of PU's to which the specified command is to be applied. Thus if the user wishes to format dozens or even hundreds of PU's, this can be accomplished readily by grouping the PU's into one or more PUG's and then specifying the PUG's after the command name. This ability to group PU's in flexible ways and then have tools process them as conceptual units is seen as one very important feature of the IST.

It should be noted that there is no prohibition against placing a single PU in several different PUG's. The user may wish to group a subroutine library together as a PUG because the library is a conceptual unit to the user. The user may then wish to group the subroutine library with several different test drivers. This can be done by creating several different PUG's which differ from each other only in that they incorporate the different test drivers. This is permitted by the IST. Furthermore, there are no diseconomies in doing so.

If, for example, the user directs that one of two overlapping PUG'S be formatted and then directs that another overlapping PUG be formatted next, the IST will recognize that the subroutine library shared by both has been formatted after having formatted the first PUG, and will not then repeat the formatting of the subroutine library in formatting the second PUG. The mechanism for effecting this efficiency will be described shortly.

The options_specification is optional. If the name of an options packet is included here, then the options specified in that packet will be used to configure the tool named in the command in processing the list of PU's. If no options packet is specified here, then the IST will access and employ a default options packet which is stored in the file system.

It is also possible for the user to specify options explicitly directly on the command line. In this case the explicitly named options are used to either augment or override the options listed in the options packet.

It is important to observe that, although the invocation of a tool through the IST may involve a great deal of work, the user is informed of the disposition of the command only by a very terse message. The purpose of this message is merely to advise the user of whether or not the command has been executed successfully, and where further information about the execution can be found. Invariably the further information will be found in a set of files, whose names will be made available to the user. Usually these files will be report files which the user may list out by using file listing commands. As observed earlier, however, it is expected that the more sophisticated analytic tools will produce report files which will be best absorbed by the user with the aid of special browsing or perusal tools. These tools will accept report files as their input and digest and format the files in response to user commands for certain types of information.

The various IST tools will create and access the various versions and views of the PU's in order to get their work done. The user will be able to access these versions and views, but will be shielded from the necessity to do so. The static data flow analysis capability, for example, will need access to a parse tree, symbol table and flow graph of all PU's of the PUG's it is directed to analyze. The user need not know any of this, however, and need only specify the name of the PUG to be analyzed. The IST command language is obliged to understand that these other files are necessary and is empowered to create them by invoking entire complex sequences of lower level tools about which the end user need know nothing. Furthermore, once these lower level tools have created these versions and views, the IST command language interpreter may choose to store them for future reuse. Thus, if the user subsequently asks to have the analyzed program formatted, the IST will recognize that some of the work needed in order to do the formatting has already been done in the process of doing earlier analysis. The IST command interpreter is equipped with sufficient logic to recognize which internal files contain this useful information and to reuse it in formatting. It is expected that these capabilities should enable the IST to effect significant efficiencies in actual use. More details about how this is accomplished will be presented shortly. In order to do so, however, is important to first understand the Virtual File System Concept.

4.4. The Virtual File System

A stated design objective for IST is that it run effectively on a wide range of machines, effectively utilizing larger amounts of storage when and if they can be made available. One way in which large amounts of storage can be effectively utilized is to store all derived and intermediate entities for possible future reuse. Storage economies can be gained by refusing to store those entities and instead regenerating them as needed. The strategy for retaining or regenerating these entities must be adjustable and transparent. It is highly desirable that both the end user and the tool ensemble always be safe in assuming that any needed named entities and derived images will always be available. Thus it is necessary that the IST file management system assume the responsibility for either retrieving these items directly or having them created or regenerated (in case storage exigencies precipitated their deletion by IST). A conceptual diagram of the virtual file system architecture is shown in Figure 4.

For example, suppose a functional tool needs the parse tree of a particular PU, call it SUBR. The requesting tool must know that SUBR`s parse tree will always be stored in a file named SUBR/TRE, and then must request it through the call:

```
CALL DBFTCH(`SUBR/TRE`, ARRAY, LEN)
```

where ARRAY is the name of the array which is to receive the parse tree, and LEN is the length of ARRAY, included to enable array overflow checking.

Subroutine DBFTCH simply looks up `SUBR/TRE` in the IST directory, accesses it and transfers it to ARRAY. It is the job of the IST command interpreter to assure that the file exists and is up-to-date. The command interpreter assures this by querying the IST directory before invoking the tool to see whether any file which the tool might need is either absent or obsolete. If so, the command interpreter sees that all such files are created. Guidance for this process comes from an internal directed acyclic graph (DAG) which specifies how the various IST file system images are derived from each other, by having each node represent a file system entity type, and each edge represent the tool needed to produce the entity at its tail from the entity at its head. Of course some tools may require and/or produce more than one file type. An example of a very simple DAG is shown in Figure 5. Using this DAG the command interpreter produces an ordered list of the steps needed to create missing files, translates this list into tool invocations, and effects their executions before invoking the original requesting tool.

In the example, the command interpreter would look up `TRE` in the dependency DAG, and see that a parse tree is derived from a token list by a parser and a token list is derived from source text by a lexical analyzer. The command interpreter would then check for the existence of the token list for SUBR. If it were present IST would cause the parser to produce the required parse tree. If the token list were absent, the command interpreter would issue commands to invoke the lexical analyzer first and the parser next. If the source text were not in the file system, an error message would be passed to the user.

This virtual strategy is also employed to enable source text versions such as formattings and instrumentations and entire static analysis and dynamic testing data bases to be purged to save space and to be recreated only on demand. This flexibility should prove useful in hosting the IST on smaller machines. Here it may be necessary to permanently store only source text. Under these circumstances all derived images and intermediate entities will be routinely purged, and always recreated on demand. This extra computation seems a reasonable trade for lack of storage.

There appears to be little experience in devising strategies for deciding which entities to delete and when to delete them for the various machines and architectures on which IST will have to be hosted. Hence a lengthy period of experimentation and adjustment will be necessary. The replacement/retention strategy will be encapsulated in a module to facilitate this.

4.5. The Command Interpreter

The IST command interpreter processor consists of two phases--compilation and sequential tool invocation (see Figure 6). Compilation, in turn, consists of three subphases: command syntactic analysis, semantic analysis, and object code generation (see Figures 7).

Syntactic analysis will, at least in later releases, be accomplished by a parser generated by a parser generator. The command language is small and uncluttered making it comfortably describable by parser- generator input. Moreover, it is recognized that users' reactions to IST may be strongly influenced by the perceived friendliness and ease of use of the command language itself. It thus seems important to enable changes in the language when and if experience indicates they are desirable. This will clearly be facilitated by the parser- generator-created parser.

The second compilation phase, semantic analysis, will be more complex entailing the selection of the standard template of IST files and functions indicated by the command (see Figure 8). In particular, the semantics of each IST command will be defined by a standard sequence of IST file system types--namely the file types which contain the information and data objects directly needed to satisfy the user's command.

Because IST employs the virtual file strategy just described, it cannot be expected that all of the files of the types which the semantic analysis phase indicates are needed will be physically present. Thus it is the job of the code generation subphase of compilation to infer from the list of needed files and the physical file system status an ordered list of intermediate files to be created and the tool fragments needed to create them.

The end product of this phase is to be a sequential file of IST code describing in detail all steps to be carried out by IST tool fragments in order to effect the specified command in the exact context of the current state of the IST file system. As such this phase might well be viewed as a pseudocompilation into a machine independent intermediate code.

In closing this description of the compilation phase, it seems important to observe that the structure of the command compiler makes it amenable to dynamic alteration so as to accept command language extensions. This is due to the fact that each of the three compilation phases is essentially table driven. The parsing phase is driven by parse tables; the semantic phase is driven by the table of templates; and the code generation phase is driven by the dependency DAG. These three tables are to be stored in the IST file system. Thus there appears to be no reason that a user tool might not be written to accept a user's specifications of how a new tool fragment is to be invoked by the user, and integrated with other fragments and file types within the file system. Such a tool is currently planned, it is expected that is will make Toolpack/IST the sort of flexible, extensible system that will grow and adapt to the needs of different user communities.

A program chaining approach such as is used by the Software Tools Project [Sche78] will most likely be used, at least in early releases to carry out the second interpreter phase--sequential tool invocation. With this strategy each tool is made into a separate main program by encasing the tool function (which is a subroutine) in a specially constructed interface main program. The sequence in which the main programs are to be executed is contained in the sequential file produced by the third phase. Effecting the indicated sequencing and error branching is the job of the interface main program. In order to carry out this job the main program must be able to access the command file, identify the next tool or tool fragment to be executed, create a command directing the invocation by the host operating system of the indicated tool or fragment's encasing main program, present the command to the host system, access the command file for the arguments needed by encased tools, and manipulate the error flags with which it will communicate with its encased tools.

Experience indicates that it is reasonable to expect all of these tasks to be readily achievable on a wide spectrum of current systems. Thus this chaining strategy seems to offer an ideal combination of flexibility and portability potential.

5. Summary

The foregoing sections have described the design and implementation strategy for the early releases of the Toolpack IST. In these sections there have also been discussions of the ways in which these releases will be used as experimental vehicles for obtaining answers to the four basic questions outlined in Section 2. Specifically:

(1) The Toolpack tool suite will be evaluated to see whether it is adequate to cover the needs of mathematical software developers.

(2) The Toolpack file system and diagnostic data bases will be instrumented and monitored to help ascertain the precise data needs of mathematical software developers as they perform their jobs.

(3) The specific tool fragments from which the Toolpack tool capabilities are constructed will be evaluated to determine how readily and flexibly they form the basis for these capabilities. Alterations to this set of fragments may be necessitated.

(4) Reactions to the proposed command language (which borrows concepts taken from Lisp, the Unix shell, and the Make processor [Feld79a]) will be monitored, and attempts will be made to determine underlying basic requirements for such languages. Perhaps more important, the adaptability of the Toolpack tool fragments and the ability to readily contrive new tools in response to changing user needs will be carefully observed. The combined efficacy of the flexible command language and the adaptable tool fragments will be studied to determine whether they effect an acceptably friendly user interface.

The tentative schedule of early releases is as follows:

Release -1: Date: November 15, 1981
Audience: Toolpack group only
Description: Command interpreter plus a few representative tools and tool fragments, some in prototype form

Release Ø: Date: September 1, 1982
Audience: Toolpack group and selected test sites
Description: Command interpreter and complete suite of tools, some may be in prototype form

Release 1: Date: Spring 1982-3
Audience: first unrestricted public release
Description: Command interpreter and complete suite of tools.

6. Future Directions and Plans

It is anticipated that the early releases of the Toolpack IST will provide definitive enough information that it will be reasonable to design with some assurance a variety of other environments which should be broader in scope.

One clear direction in which these techniques should be readily extended is towards newer and more interesting languages. A frequent criticism of this work is that it is too centered on Fortran, a language which is becoming increasingly out of step with the most modern thoughts about high level languages. It seems inappropriate to debate the relative merits of Fortran here, especially since this paper should, by now, have made it clear that Fortran was chosen primarily because it offers an excellent experimental vehicle. A critical mass of Fortran tools is already in existence and a receptive, perceptive community of users of both the language and tools is also in existence.

Of greater importance, however, is that the IST design concepts and actual software have been designed in such a way as to facilitate their conversion to support other languages. Lexical analysis, syntactic analysis and semantic analysis have all been modularized in separate tool fragments. Other tool fragments operate on abstract representations of source programs, and are thus well insulated from the source language and its peculiarities. It seems that the IST could offer similarly strong support for other languages is the lexical, syntactic and semantic analysis tool fragments were replaced by tool fragments for analyzing some other language. Moreover, the lexical and syntactic analyzers are currently automatically produced from analyzer generators. Some research is underway to investigate the practicality of using attribute grammars to specify and drive the creation of semantic analyzers as well. Thus it appears that there is hope that these three language dependent front-end tool fragments might one day be automatically generated from a language specification.

This seems to be a good plan for the future, and there is honest optimism that it is an achievable goal. This optimism must, however, be tempered by the suspicion that many of the back-end tool fragments incorporate implicit assumptions about the semantics of Fortran which will only be ferreted out and concentrated in the semantic analysis module after considerable experimentation, observation, and adjustment.

It would seem that the best way to work in this direction is to promptly begin design of an IST-like capability for a more modern, sophisticated and challenging language. Ada [3] seems to a be a very logical choice. One reason for this choice is that, here too, there is a community that is sensitive to the importance of tools. The Ada community has in fact already gone to the trouble of producing a preliminary specification for a tools environment [Buxt 80]. Prototype environments for Ada are already under construction. They are not based upon the notions of reusable tool fragments and a virtual data base, and thus should serve as illuminating comparison bases with an environment which is based upon these notions.

Ada also seems a good choice because of the ways in which it encourages and supports the concept of modularity. There are language facilities for writing higher level code which assumes the existence of support modules whose code may not be visible. What is visible to the writer of this higher level code is a summary view of the interface which this support code presents. Clearly this summary view is a very logical candidate for inclusion in a file system supporting the construction of the higher level code. The Ada language itself does not go beyond the point of mandating that a rudimentary view of this interface be furnished by the creator of the support modules. It is clear, however, that such a rudimentary view is minimal, but far less than desirable. For example it would be preferable to have this view contain enough information to enable the sort of thorough data flow analysis that is suggested in [Tayl 80] and [Tayl 80a]. The writer should not be burdened with having to provide such a large amount of information, however.

[3]Ada is a trademark of the U.S. Department of Defense Ada Joint Program Office.

With an architecture such as this paper suggests, however, the information could be created by analytic tool fragments and stored in the file system for future use. A large body of diverse information about support modules could be accumulated incrementally over a long period of time and either stored over a period of time (or regenerated on demand according to the dictates of local economics) for the benefit of users who wish to be guided to the proper use of these modules.

It is worthwhile to observe in closing this discussion of application to other languages that the language in which tool fragments is coded should not be a concern of the end user. In particular there is no apparent reason that the language in which tool fragments are coded be the same as the language in which the code being analyzed is written. Toolpack/IST tool fragments are currently written primarily in languages such as Ratfor which are converted by preprocessors into Fortran 66. This was done because Fortran 66 was perceived as being the most reliable portability vehicle, and one of the requirements of the Toolpack project is that Toolpack software be portable. Other languages, such as Pascal are becoming sufficiently standardized that they can no longer be overlooked as portability vehicles. Other environments, moreover, may not need to be as highly portable. Under these new and changing circumstances it would be reasonable to code tool fragments in other languages. These tool fragments could coexist quite nicely with older tool fragments written in Fortran derivative languages if that were to prove to be expedient.

Another important issue that will be addressed as an outgrowth of this work is the issue of how best to distribute this sort of environment across a range of newer hardware, such as personal work stations. This issue clearly comes up as a consequence of viewing the central file system as being multiuser accessible. If the various users are considered to be working at their own work stations, then the central file system is already distributed. A whole range of contingent issues then become important.

Finally, there is the issue of going beyond the current user interface language with its tool orientation to a language which is knowledge based and oriented towards being a query system. It appears that current and contemplated user interfaces are too tool oriented, requiring users to be at least somewhat expert in the capabilities of individual tools. It would be far preferable if the user interface was designed so as to give users the impression of the existence of a knowledge base. Thus the development environment's user interface would take on more the appearance of a query system to support program development than an imperative language for invoking tools. The Toolpack architecture leans in this direction because of the virtual strategy adopted for the file system. Needed files are automatically materialized in indirect response to tool invocations. It is only an extension of this notion to consider that tools themselves might be automatically invoked as well in indirect response to requests for key program development information. The experimental results gotten from early releases should indicate how far this inclination should be continued, as well as the ramifications of such a continuation for the internal architecture.

7. ACKNOWLEDGMENTS

The design of the Toolpack IST is the product of a great deal of collaborative work and discussions with individuals inside and out of the project. In particular the original design concepts of the command language were contributed by Stuart I. Feldman of Bell Telephone Laboratories. The concepts underlying his Make processor were, obviously, also quite influential in the design of IST.

The implementation work has been ably led by Allan L. Shafton. The design and implementation of key command interpreter tool fragments was led by Geoff Clemm, Brent Welch and Mark Maybee.

Finally the financial support of the National Science Foundation, Department of Energy and U.S. Army Research Office is gratefully acknowledged.

8. REFERENCES

[Bake 77] Baker, B.S., "An Algorithm for Structuring Flowgraphs" Journal of the ACM, 24 #1 pp. 98-12Ø (January 1977).

[Boyl 76] J. M. Boyle and K. Matz, "Automating Multiple Program Realizations," MRI conf. Rec. XXIV Symp. on Computer Software, Polytechnic Press, Brooklyn, N. Y., pp. 421-456 (1976).

[Buxt 8Ø] J. N. Buxton, V. Stenning, "Requirements for Ada programming support environments," Stoneman, Department of Defense (February 198Ø).

[Clem 79] G. M. Clemm, "CLEMSW User's Manual," Tech. Report #CU-CS-167-79, Dept. of Computer

Science, University of Colo., Boulder, Colo., 1979.

[Clem 81] G. M. Clemm, "FSCAN-81 Report and User's Manual," Tech. Report #CU-CS-2Ø2-81, Dept. of Computer Science, University of Colo., Boulder, Colo., 1981.

[Cowe 77] W. R. Cowell, L. D. Fosdick, "Mathematical Software Production," in Mathematical Software III, Academic Press, N. Y., pp. 195-224, 1977.

[Donz 8Ø] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The Mentor Experience," INRIA Research Report No. 26, INRIA, Rocquencourt, France, 198Ø.

[Dorr 74] J. Dorrenbacker, D. Paddock, D. Wisneski and L. D. Fosdick, "POLISH, A Fortran Program to Edit Fortran Programs," Tech. Report #CU-CS-Ø5Ø-74, Dept. of Computer Science, University of Colo., Boulder, Colo., 1974.

[Feib 81] J. Feiber, R. N. Taylor, L. J. Osterweil, "Newton--A Dynamic Program Analysis Tool Capabilities Specification," Tech. Report #CU-CS-2ØØ-81, Dept. of Computer Science, University of Colo., Boulder, Colo., March 1981.

[Feld 79] S. I. Feldman "The Programming Language EFL," Computing Science Tech. Rpt. #78, Bell Laboratories, Murray Hill, New Jersey, June, 1979.

[Feld 79a] S. I. Feldman, "Make--A Program for Maintaining Computer Programs," Software-Practice and Experience 9 (April 1979), pp. 255-265.

[Fosd 81] L. D. Fosdick, "POLISH-X Transformations," University of Colorado, Dept. of Computer Science, Tech. Report #CU-CS-2Ø3-81 (May 1981).

[Hague 81] S. J. Hague, "The Provision of Editors for the Manipulation of Fortran," Toolpack Document SJH 11112 (November 1981). Available from Applied Mathematics Division, Argonne National Laboratory, Argonne, Ill. 6Ø439.

[Hans8Øa] D. R. Hanson, "A Portable File Directory System," Software Practice and Experience 1Ø (August 198Ø), pp. 623-634.

[Hans 80b] D. R. Hanson, "The Portable I/O System PIOS," University of Arizona, Dept. of Computer Science, Tech. Report #80-6a (April 1980, revised December 1980).

[John 75] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," Bell Laboratories Computing Science Tech. Rpt. #32 (1975).

[JPL 78] Proceedings of Conference on the Programming Environment for Developing Numerical Software, Jet Propulsion Laboratory, Pasadena, Calif., Oct. 18-20, 1978.

[JPL 81] Proceedings of Conference on the Computing Environment for Mathematical Software, Jet Propulsion Laboratory, Pasadena, Calif., July 15, 1981.

[JPL 81a] SFTRAN III, Programmer Reference Manual, JPL Internal Document, 1846-98, Pasadena, Calif., April 1981.

[Kern 75] B. W. Kernighan, "Ratfor--A Preprocessor for a Rational Fortran," Bell Laboratories Computing Science Technical Report #55.

[Meyers 81] E. W. Meyers, Jr., and L. J. Osterweil, "BIG-MAC II: A Fortran Language Augmentation Tool," Proc. 5th Int'l Conf. on Software Eng., IEEE Cat# 81CH 1627-9, pp. 410-421, (March 1981).

[Oste 76] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation, Error Detection, and Documentation System for FORTRAN Programs," Software - Practice and Experiences 6 pp. 473-486 (Sept. 1976).

[Oste 81] L. J. Osterweil, "Software Environment Research Directions for the Next Five Years," Computer 14 pp. 35-43, (April 1981).

[Oste81a] L. J. Osterweil, "Draft Toolpack Architectural Design," Dept. of Computer Science, Univ. of Colorado at Boulder, Nov. 1, 1981.

[Ryde 74] B. G. Ryder, "The PFORT Verifier," Software - Practice and Experience," 4 pp. 359-377, (1974).

[Sche 78] Scherrer, D., COOKBOOK, instructions for implementing the LBL software tools package.

Internal Rep. LBID098, Lawrence Berkeley Laboratory Univ. of California, Berkeley, 1978.

[Tayl 80] R. N. Taylor, L. J. Osterweil, "Anomaly detection in concurrent software by static data flow analysis, IEEE-Transactions on Software Engineering, SE-6, 3 (May 1980), pp. 265-278.

[Tayl 81] R. N. Taylor, "Static Analysis of the Synchronization Structure of Concurrent Programs, Ph.D. Thesis, Dept. of Computer Science, University of Colorado, Boulder, Colorado, 1980.

[Teit 81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Communications of the ACM 24 (September 1981) 563-573.

[Ward 81] W. Ward and J. Rice, "A Simple Macro Processor," Toolpack Document WW/JR 10921 (September 1981). Available from Applied Mathematics Division, Argonne National Laboratory, Argonne, Ill. 60439.

[Weth 81] C. Wetherell and A. Shannon, "LR-Automatic Parser Generator and LR(1) parser," IEEE Trans. on Software Eng. SE-7, pp. 274-278, (May 1981).

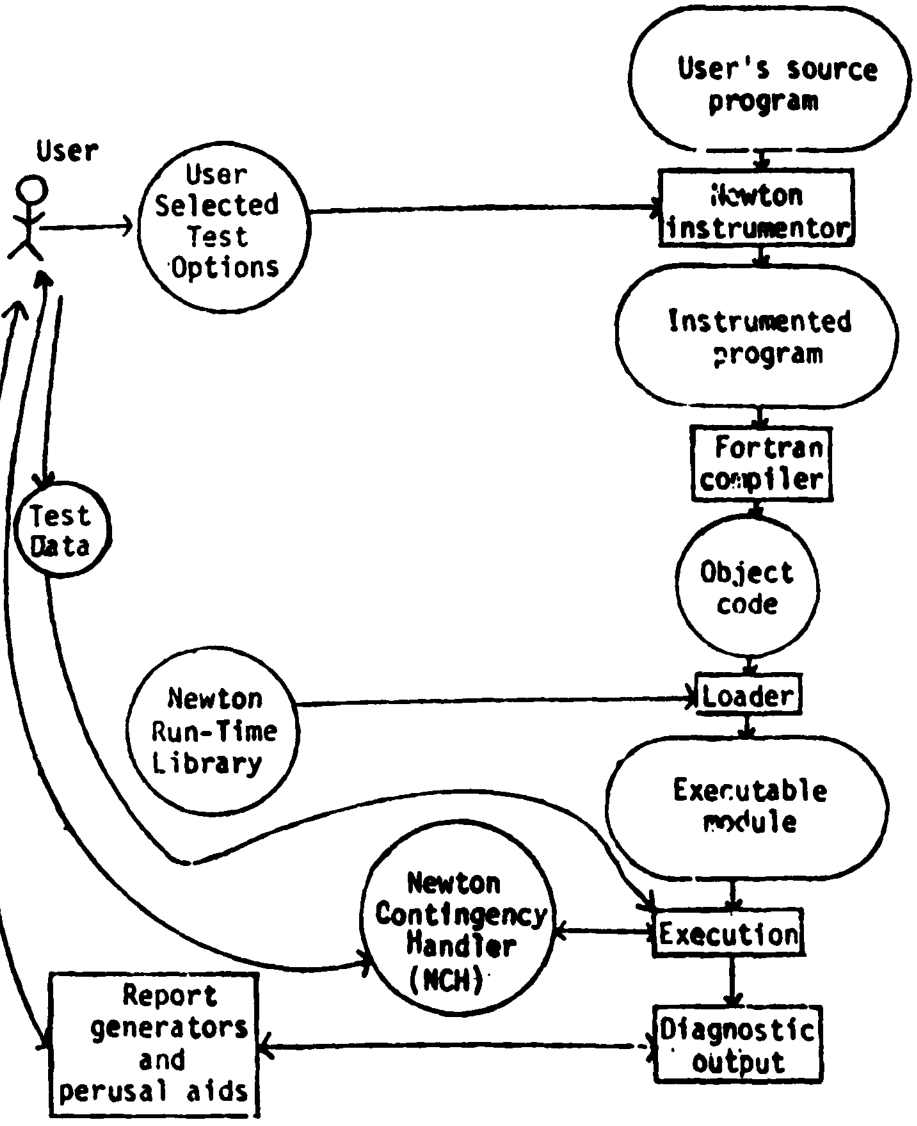


Figure 1: Use of Dynamic Test Tools

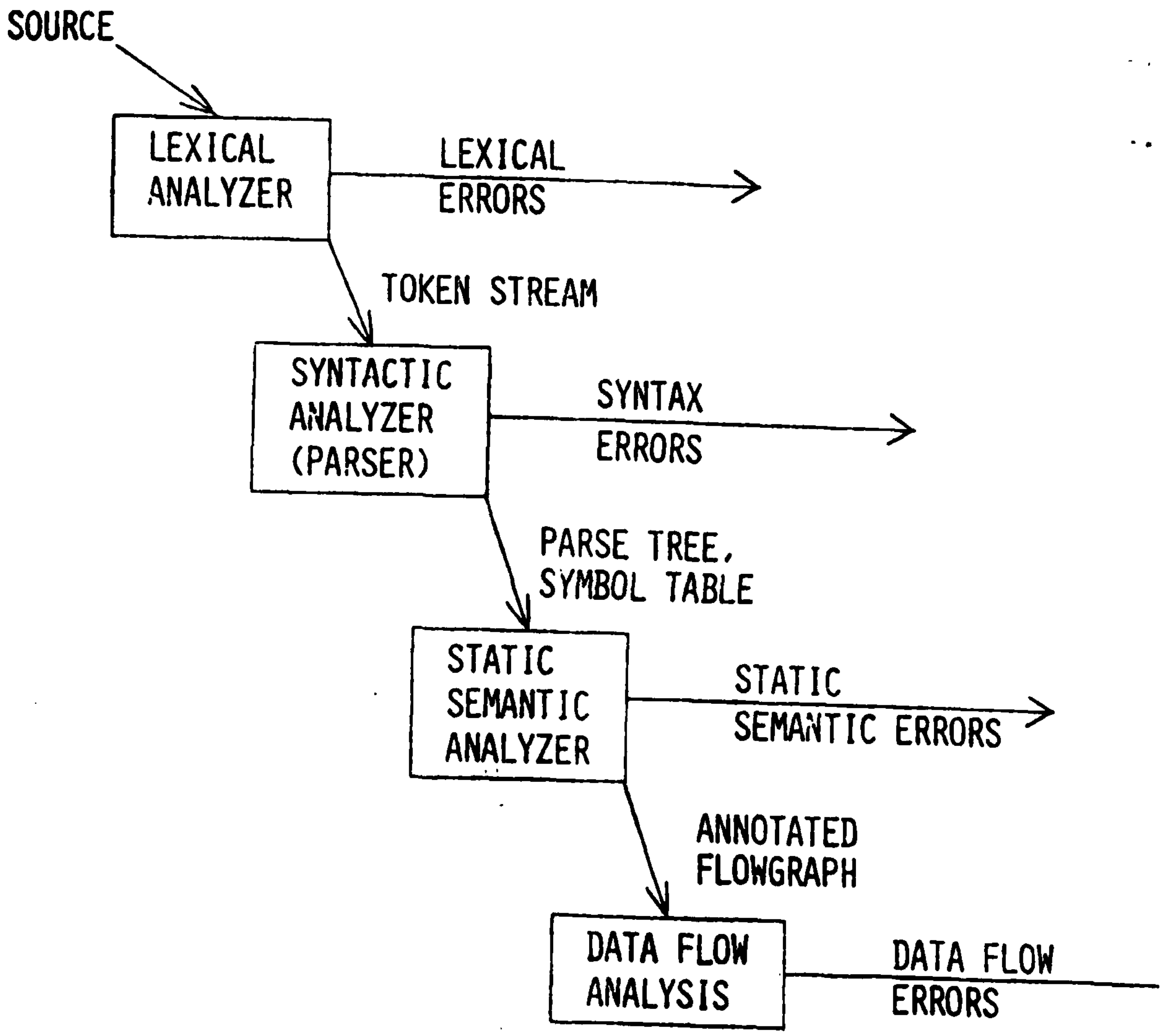

SOURCE
LEXICAL ANALYZER
LEXICAL ERRORS
TOKEN STREAM
SYNTACTIC ANALYZER (PARSER)
SYNTAX ERRORS
PARSE TREE, SYMBOL TABLE
STATIC SEMANTIC ANALYZER
STATIC SEMANTIC ERRORS
ANNOTATED FLOWGRAPH
DATA FLOW ANALYSIS
DATA FLOW ERRORS

Figure 2

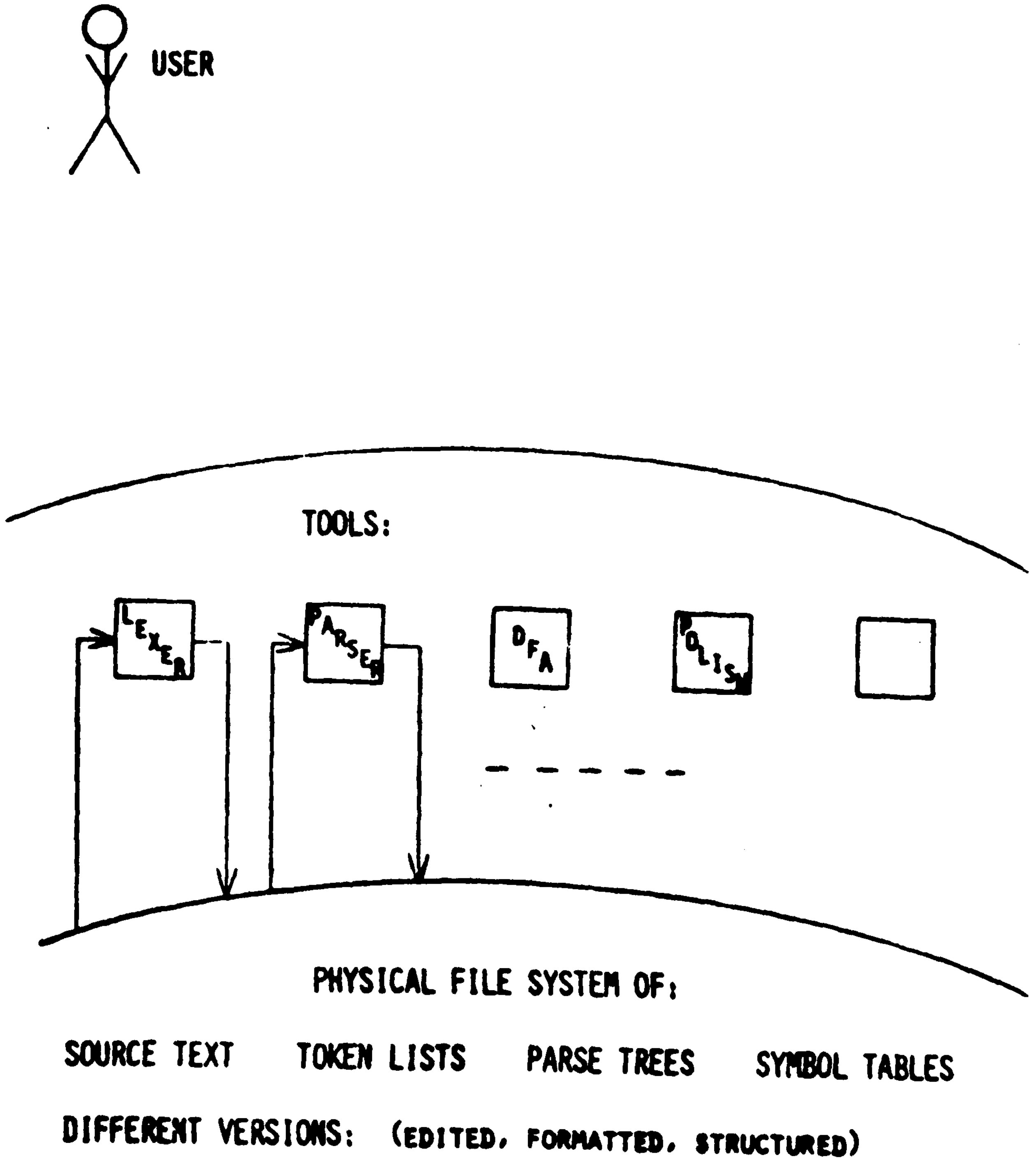
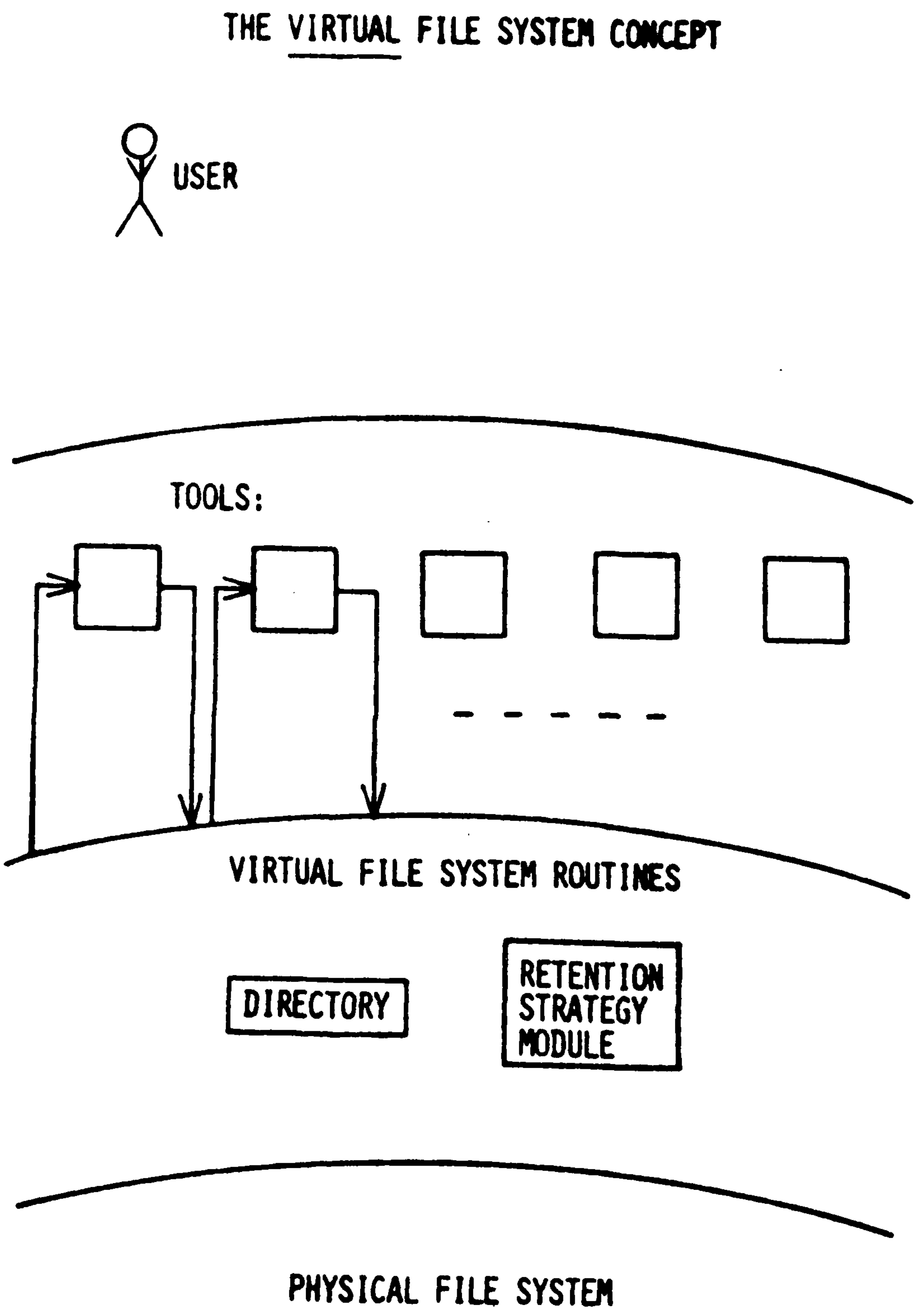

USER
TOOLS:
LEXER
PARSER
DFA
POLISH
PHYSICAL FILE SYSTEM OF:
SOURCE TEXT
TOKEN LISTS
PARSE TREES
SYMBOL TABLES
DIFFERENT VERSIONS: (EDITED, FORMATTED, STRUCTURED)

Figure 3


THE VIRTUAL FILE SYSTEM CONCEPT
USER
TOOLS:
VIRTUAL FILE SYSTEM ROUTINES
DIRECTORY
RETENTION STRATEGY MODULE
PHYSICAL FILE SYSTEM

ENTITIES

Figure 4

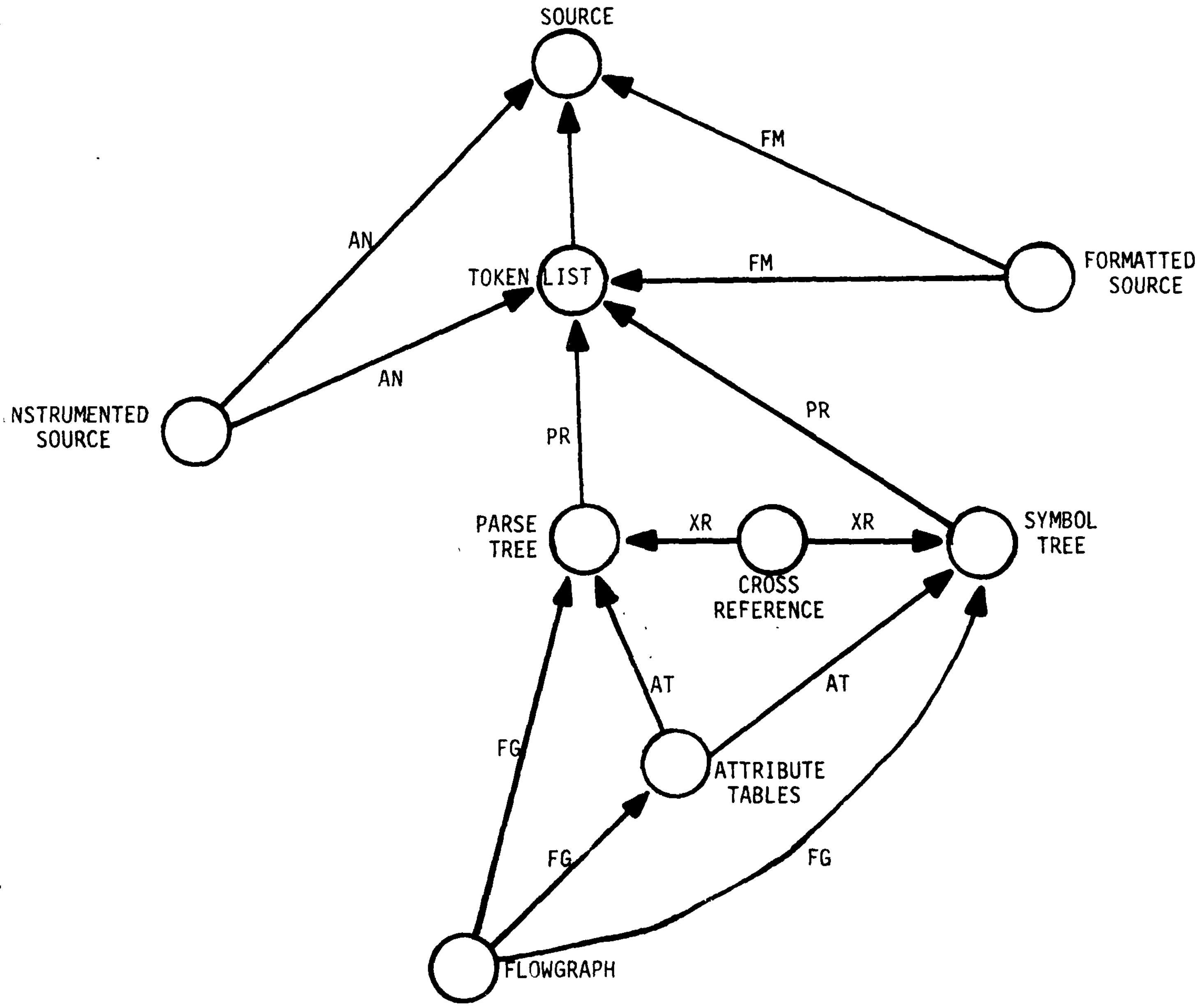


Figure 5: A sample dependency DAG. Each node represents a type of file to be found in the Toolpack/IST file system. Each edge represents the dependency of this file type on another file type. Specifically, the file type at the head may be required in creating the file at the tail. The label on the edge represents the tool fragment needed to create the file type at the tail. Note that some tool fragments (eg. FM) require more than one file type to produce their output, while others (PR) produce more than one file type.

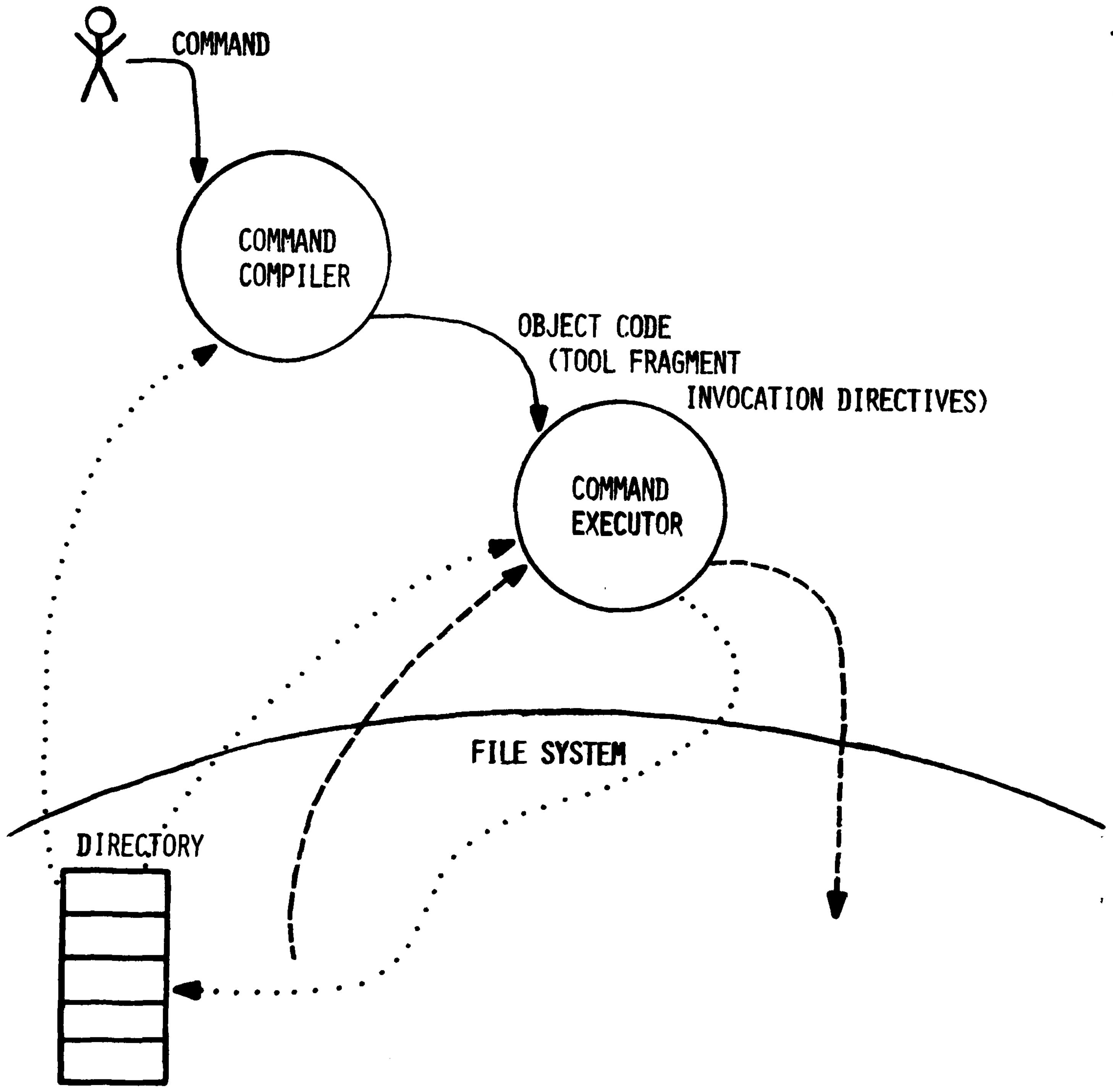
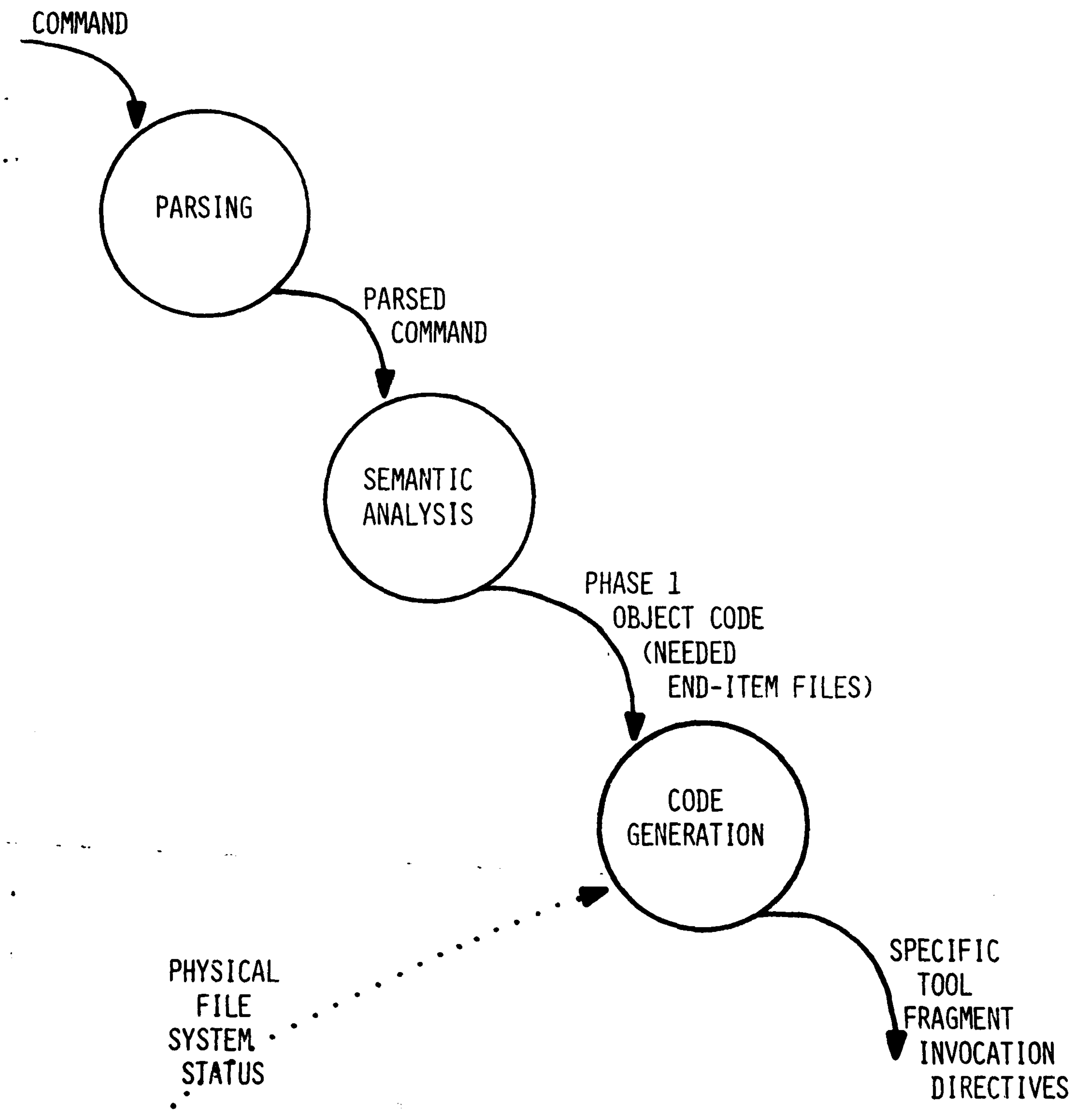


Figure 6: Overview of the IST Command Interpreter showing two main phases--compilation and execution. Dotted lines show flow of information into and out of the file system directory. Dashed lines indicate data flow thorugh the file system itself.



COMMAND
PARSING
PARSED COMMAND
SEMANTIC ANALYSIS
PHASE 1 OBJECT CODE (NEEDED END-ITEM FILES)
CODE GENERATION
PHYSICAL FILE SYSTEM STATUS
SPECIFIC TOOL FRAGMENT INVOCATION DIRECTIVES

Figure 7: Breakdown of the Command Compilation process into its three subphases. Note that only code generation requires information about current file system status.

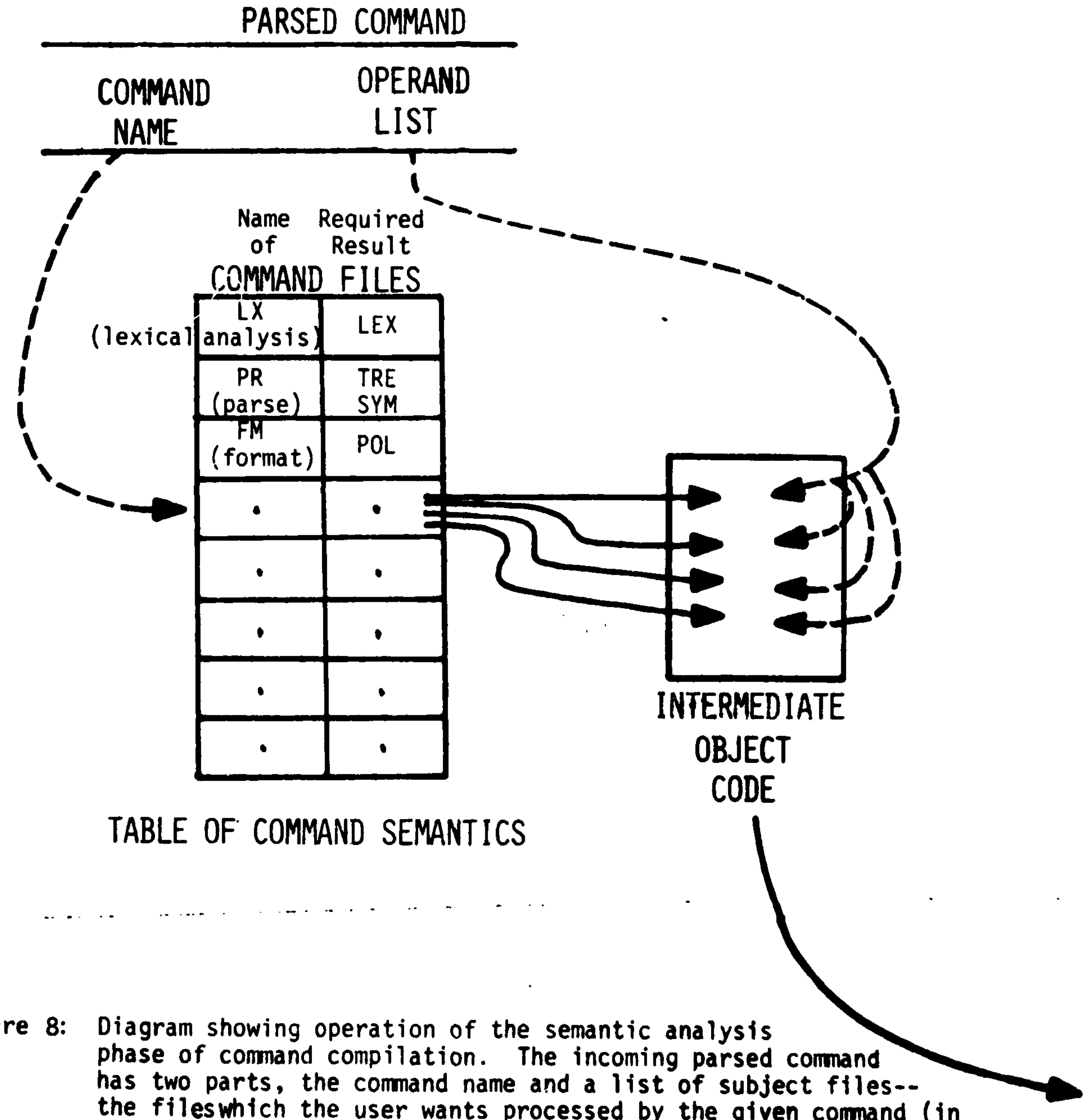


Figure 8: Diagram showing operation of the semantic analysis phase of command compilation. The incoming parsed command has two parts, the command name and a list of subject files-- the fileswhich the user wants processed by the given command (in this example no options are specified). In this phase, the command name is used as an index into the Table of Command Semantics which associates with the command name, the names of one or more file types which contain the information which the user is requesting in invoking the stated command. Intermediate Object Code is formed by pairing off all combinations of such file types and stated operand files.